International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR)
ISSN(P): 2249-6831; ISSN(E): 2249-7943
Vol. 7, Issue 1, Feb 2017, 35-52




# MEDIAN FIRST TOURNAMENT SORT

**MUKESH SONI & DHIRENDRA PRATAP SINGH**

*Department of Computer Science & Engineering, Maulana Azad National Institute of Technology, Madhya Pradesh, India*

***ABSTRACT***

***Rearrangement of things for making it more usable needs a mechanism to do it with a practically possible time. Sorting techniques are most frequently used mechanism used for rearrangement. Selection of a procedure to sort something plays vital role when the amount to be sorted is large. So the procedure must be efficient in both time and space when it comes to the matter of implementation. Several sorting algorithms were proposed in past few decades and some of them like quick sort, merge sort, heap sort etc. widely using in real life scenarios since they found their way to be top on the list of sorting mechanisms ever proposed. This paper proposing a Median first Tournament Sort which practically beats quick, merge, and heap sorting mechanisms in terms of time and number of swaps required to sort.***

***KEYWORDS: Sorting, Median, Tournament tree, Merge Sort & Heap Sort***

**Received:** Dec 07, 2016; **Accepted:** Jan 25, 2017; **Published:** Feb 01, 2017; **Paper Id.:** IJCSEITRFEB20175

Original Article

## I. INTRODUCTION

To process data collections like integer list of elements or a list of string (names), computers are used widely and a great portion of its time is taken away by maintaining the data to be processed in sorted order in the first place. The basic concept behind sorting a list of elements is that they start out in some random order and need to be arranged from lowest to highest or highest to lowest.

Sorting is the most fundamental algorithmic problem, which can be defined as taking an arbitrary permutation of *n* number of items and rearranging them into a total order. This order can be ascending or descending as shown in equation 1 and 2 respectively.

$$xi \geq xj, for\ i > j \quad \textbf{(1)}$$

$$xi \leq xj, for\ i > j \quad \textbf{(2)}$$

Importance of sorting algorithms can be conceived from the following case: while searching in a data space, 25% of all CPU cycles are spent sorting; this lead to different approaches useful in sorting, and these ideas can be used to solve many other problems such as operations research [1], numerical computations [2], broadcast communications [3, 4] and various others like searching, closet pair, Element uniqueness, Frequency distribution, selection, convex hull [5, 6, 7].

If the number of items to be sorted is less, a human reader may be able to arrange these items within a glance of second. If the number of items is in order of few hundred or more, then a more systematic approach is required. Mostly, sorting is used in combination with searching for making the traversal process more elegant.

Sorting algorithms are classified in two categories: comparison based sorting algorithms and non-comparison based sorting algorithms; comparison sorts being much popular.

A comparison based sorting algorithm reads the list of elements through a single abstract comparison operation (like greater than), this determines which of two elements being compared should occur first in the final sorted list. There are many algorithms available in literature under this category: like Bubble sort [11] [12], Selection sort [14][15], Insertion sort [16], Shell sort [17], Merge sort [18], Heap sort [13], Quick sort [19].Non-comparison based sorting algorithms are: Bucket sort, Radix sort (MSD and LSD) [8], Bit Index Sort [9].

This paper presents anew comparison based sorting algorithm that uses tournament trees [10], in which elements are sorted by partitioning the list in two sets with the help of median elements.

The rest of the paper is organized as follows. In Section II, literature review and discussion of several comparison based sorting algorithms is presented. Section III defines the proposed algorithm, section IV discuss running complexity and validations. In Section V, numerical example is explained in section VI experimentation results are shown and the inferences made from it. Section VII, finally summarizes the paper.

## II. LITERATURE REVIEW

Several comparison based sorting algorithms have been proposed previously, this section gives a brief review of some of these existing sorting algorithms.

### Bubble Sort

This is the simplest algorithm which checks whether current element is greater than next element or not; if this condition holds true, elements are swapped and larger element is moved to the end of the list. Each iteration of Bubble sort (BS) gives one maximum element from the remaining unsorted $k$ elements of $n$ elements list, where $(n - k)$ larger elements already have been placed at its correct position. During execution of any $i^{th}$ it eration, algorithm requires $(n - i)$ comparisons.

Let $C_n$ denotes the number of comparisons required to sort $n$ elements, then its value is given in equation (3.1).

$$C_n = \sum_{j=1}^{n}(n - j) = O(n^2) \quad \textbf{(3.1)}$$

Assume $S_n$ denotes the number of swaps required by Bubble sort algorithm, when all elements are sorted in decreasing order then $k^{th}$ pass of Bubble sort requires $(n-k)$ swaps, so total swap operations required is derived in equation (4).

$$S_n = \sum_{j=1}^{n}(n - j) = O(n^2) \quad \textbf{(3.2)}$$

Bubble sort algorithm requires $O(n^2)$ comparisons and $O(n^2)$ swaps on input size of $n$. Bubble sort is the oldest and the simplest sorting algorithm. The algorithm gets its name from the way smaller elements are bubbled to top of the list [11]. This algorithm is not considered efficient and not used anywhere except for theoretical ideas [12].

### Heap Sort

Heap sort algorithm builds heap data structure from the given elements to be sorted. Heap data structure can be of two types: min heap and max heap. In max heap data structure, each node value is greater than the node value of its children [13]. Heap can be built in $O(n)$time complexity using Build Heap method. The cost needed to build heap differs with height of the tree $h$and the number of elements present at that height. Number of elements present in heap at height $h$ is given in

equation (4.1). Let the number of elements at height *h* is denoted by $NE_h$.

$$NE_h = \left\lceil \frac{n}{2^{h+1}} \right\rceil \quad \textbf{(4.1)}$$

Cost of building a heap is given in equation (4.2) which is derived to be *O(n)*.

$$\sum_{h=0}^{\lfloor log\ n \rfloor} NE_h\ \ O(h) = O(n) \quad \textbf{(4.2)}$$

After building the heap, root element is deleted and removed from it, now from remaining *(n – 1)* elements heap is built again and this process continues until all elements are deleted. Each element deletion takes *O(log(n))* time complexity, and this process is repeated*(n – 1)*times to sort all elements. So, total time complexity of Heap sort comes out to be *O((n-1)log(n)) = O((n)log(n))*.It is one of the optimal comparison based sort algorithm [3] and is a variation of Selection sort, in which *O(n)* comparisons are required to find the smallest element, however if heap is built it can be done in *O(log n)* time complexity.

**Selection Sort**

Selection sort is a well-known sorting algorithm which works by finding the smallest element in each pass. In a pass, selection sort finds a single smallest element among the *k* remaining unsorted elements of *n* elements list where *(n - k)* elements are sorted. Selection sort algorithm compares two elements at once to find the smallest between them and proceed to update the smallest if found any further. Selection sort takes *n(n-1)/2* comparisons, however number of swaps are in *O(n)*. Selection sort algorithm is considered best in terms of number of swaps required because in each pass exactly a single swap is required if needed. Selection sort's performance is neither good for worst case nor for average case input sequence. For longer lists, insertion sort performs better than selection sort [14]. Selection sort is having property that its running time does not vary by the order of size of input elements list [15].

**Insertion Sort**

Insertion sort is an efficient algorithm for sorting a small list of elements. It works in the same way as people sorts a hand of playing cards. We have an empty left hand and cards are on table. Then remove one card at a time from the table and insert it into its correct position on the left hand. To find the correct position, compare the current card with cards on the left hand; for comparison either linear search or binary search can be used. At all the times cards on left hand remains sorted. Let $T_n$ be the running time of algorithm for *n* elements. Then worst case running time of insertion sort is given in equation (5).

$$T_n = an^2 + bn + c \quad \textbf{(5)}$$

where, a, b, and c are constants which depends on individual statements cost. Best case performance of insertion sort is in linear order of input size. When size of the list is very large then it is slower [16].Both number of comparisons and swaps required in average case for insertion sort is $((n^2/4) + \Theta(n))$.

**Shell Sort**

This algorithm partitions the list of numbers into noncontiguous sub list having elements of some step size *h* apart. Each sub list is sorted by applying Insertion sort algorithm and in each pass step size is decreased until it becomes *1* and by the end of it, the list is sorted. The important issue in shell sort is of finding the optimal step size *h* which cannot be predicted. However, Knuth [17] have shown that if step size is chosen either $(16n/\pi)^{1/3}$or *1*, then shell sort takes $O(n^{3/2})$. Step size can be chosen by the following rule given in equation (6).

$$\begin{cases} h_1 = 1 \\ h_{i+1} = 3 * h_i + 1 \ \forall \ i \geq 1 \end{cases} \quad \textbf{(6)}$$

**Merge Sort**

In Merge sort, the list is chopped into two or more sub list that are of nearly equal size. Sort each sub list individually and then merge the resulting sorted sub list to achieve the final sorted list. Merge sort's performance depends on two activities; partitioning and merging. The number of partitioning steps is equal to the number of merging steps. In merge sort, the input list is divided by factor of *2* so at most⌈*(log n)*⌉ levels are required in recursion tree. It shows that during merging at most⌈*(log n)*⌉passes are required; in each pass, every list element might be used at most once for a comparison, therefore *n* comparisons per pass are needed. Hence, total number of comparisons needed for Merge sort is*Θ(n*⌈*(log n)*⌉*)*. Merge sort is an optimal comparison based sorting algorithm. Merge sort is an out-place sorting algorithm; it requires extra space of *O(n)*, where *n* is number of elements in the list [18].

**Quick Sort**

Quick sort is similar to merge sort; it is a divide and conquer approach, and involves successive partitioning of the list to be sorted. The main difference lies in the partitioning of the list and sub lists are maintained in proper relative order and hence merging is not required. Quick sort selects a pivot element to partition the list into two sub list, in which one sub list contains elements which are less than the pivot element and the other list contains elements that are greater than the pivot element. This process is repeated recursively for each sub list until every sub list contains at most one element. Total cost of partitioning in worst case is *Θ(n)*. Let $C_n$ be the number of comparisons and $S_n$ be the number of swaps required by quick sort when applied to a list of *n* elements. Let partitioning produce one sub list of *d* elements and another sub list of *(n – d – 1)* elements, then equation (7.1) holds:

$$C_n = C_d + C_{(n-d-1)} + n - 1 \quad \textbf{(7.1)}$$

As the initial partitioning require *(n –1)* comparisons. In worst case, the number of comparisons required by quick sort algorithm is given in equation (7.2) after solving equation (7.1):

$$C_n = 0.5n^2 - 0.5n \quad \textbf{(7.2)}$$

Similarly, the number of swaps required in worst case is given in equation (7.3):

$$C_n = 0.5n^2 - 1.5n - 1 \quad \textbf{(7.3)}$$

Equation (7.2) and (7.3) shows that quick sort is as good as selection sort in worst case [19].

## III. PROPOSED ALGORITHM

Before discussing the proposed algorithm some definitions needs to be addressed:

- **Definition 1:** Rank of a player(r(x)): Assuming a set of N players participating in a tournament, and by the end of it each will have a unique rank position based on the number of players each can beat. These N players are then ranked from 1 to N. So, to get the rank of a player "x", we call a function "r(x)"that returns a natural number greater than equals to 1and smaller than equals to N.
- **Definition 2:** Winner Set ($N_t$) and Loser Set ($N_b$): Let N be the set of all players in the tournament, $N_t$ be the set of N/2 top ranked players and $N_b$ be the set of N/2 bottom ranked players. Then equation (8) holds.

$\forall x \in N_t, \forall y \in N_b => r(x) < r(y)$

where, $|N_t| = |N_b| = N/2$ **(8)**

- **Definition 3** -Rank Relation (R): R relation is said to be "player of minimum rank" defined on N×N and satisfies following conditions:

$x_t \in N_t$

$x_b \in N_b$

$N_t \cap N_b = \phi$

$|N_t| = |N_b| = N/2$

$R(x_t, x_b) = x_t, [since, r(x_t) < r(x_b)]$

**Definition 4***: Balanced tournament:* The fixture or tournament is said to be balanced if and only if equation (8) holds.

**Definition 5***: Imbalanced tournament:* The fixture or tournament is said to be imbalanced if and only if equation (9) holds.

$\exists x \in N_t, \exists y \in N_b => r(x) > r(y)$ **(9)**

The proposed algorithm called Median First Tournament Sort (MFTS) uses the concept of Tournament trees [9] to sort the given list. In this algorithm, players are divided into two sets; one set is for the half of all the players ranked top in the tournament called *winner set* and the other set contains remaining half of the players ranked at bottom half called the *loser set*, so the number of players will be equal in both the sets. To divide players in to two mentioned sets, each player compete with any one of the remaining players. The winner between them goes to the winner set and the loser goes to the loser set. Here we assume every player of the *winner set* can beat all players of the *loser set*. If any player from *winner set* cannot beat anyone of the player of *loser set*, then the fixture was not balanced and swapping of some players from *winner set* to *loser set* and from *loser set* to *winner set* should be done.

For swapping of the players from one set to another, a player from *winner set* and a player from *loser set* are chosen such that the player of *loser set* beats the loser player of *winner set*. These are the ideal choices to be swapped. If no such biggest loser player exists in *winner set* such that it can't be beaten by the biggest winner of the *loser set* then the fixture is said to be balanced. Biggest loser is defined as the player of the set that cannot beat anyone else in the same set and biggest winner is defined as the player of the set that can beat everyone else in the same set.

If the fixture is balanced then the biggest loser of *winner set* say "$P_{n/2}$" and biggest winner of *loser set* say "$P_{n/2+1}$" are the middle ranked players (i.e. median players). So the position of these players is now fixed. Now removing these players from their corresponding set, we again find the biggest loser in the *winner set* say "$P_{n/2-1}$" and biggest winner in the *loser set* say "$P_{n/2+2}$", original position of "$P_{n/2-1}$" player is immediately before the position of "$P_{n/2}$" player and original position of "$P_{n/2+2}$" player is immediately after the position of "$P_{n/2+1}$". Continuing in this way all the players will be placed at their respective positions according to their ranks and hence sorted.

Elements to be sorted are considered as players each having some rank on the basis of which they should be arranged in ascending order. On the basis of this rank, winner and loser are decided in a match such that winner has lower

rank number and loser has higher rank number. If an element exists such that it has same rank as another element then the element that comes first in the list is considered to be a winner to ensure the stability of the algorithm.

**Algorithm 1:** CREATE_TOUR_TREE (ARR[], N)

**Input:** ARR[] is the array for which tournament tree to be generated, N is the size of the array ARR[]

**Output:** Tournament Tree Array B[2][N - 1]

1. Define **B[2][N - 1],I := (N – 1) / 2,J := (N – 1) / 2,K := 0**
2. WHILE **K < N** DO
   - IF **A[K] <= A[K + 1]** THEN
     - **B[0][I++] := A[K]**
     - **B[1][J++] := A[K + 1]**
   - **ELSE**
     - **B[0][I++] := A[K + 1]**
     - **B[1][J++] := A[K]**
   - **END IF**
   - **K := K + 2**
3. END WHILE
4. **I := ((N – 1) / 2) – 1**
5. WHILE **I > -1** DO
   - IF **B[0][2 * I + 1] > B[0][2 * I + 2]** THEN
     - **B[0][I] := B[0][2 * I + 1]**
   - **ELSE**
     - **B[0][I] := B[0][2 * I + 2]**
   - END IF
   - IF **B[1][2 * I + 1] < B[1][2 * I + 2]** THEN
     - **B[1][I] := B[1][2 * I + 1]**
   - ELSE
     - **B[1][I] := B[1][2 * I + 2]**
   - END IF
   - **I = I - 1**

6. END WHILE
7. Return B[][]

**Algorithm 2:** UPDATE (B[][], N, TYPE)

**Input:** B[][] is the tournament tree array which needs to be updated, N is the size of the array B[][]

**Output:** Updated Tournament Tree Array B[2][N - 1]

1. Define **I := 0, J := 0, FLAG1 := 0, FLAG2 := 0**
2. Define **E1 := B[0][0], E2 := B[1][0]**
3. WHILE **2 * I + 2 < N – 1** DO
   - IF **B[0][2 * I + 1] = E1** THEN
     - **I := 2 * I + 1**
   - **ELSE**
     - **I : = 2 * I + 2**
   - **END IF**
4. END WHILE
5. WHILE **2 * J + 2 < N – 1** DO
   - IF **B[1][2 * J + 1] = E2** THEN
     - **J := 2 * J + 1**
   - **ELSE**
     - **J : = 2 * J + 2**
   - **END IF**
6. END WHILE
7. IF **TYPE = UPDATE** THEN
   - **B[0][I] := MIN_INF** //Replace with Negative Infinity
   - **B[1][J] := MAX_INF** //Replace with Positive Infinity
8. ELSE IF **TYPE = SWAP** THEN
   - **B[0][I] := E2**
   - **B[1][J] := E1**
9. END IF
10. WHILE **FLAG1 + FLAG2 < 2** DO

- IF **I % 2 = 1 AND FLAG1 != 1** THEN
  - IF **B[0][I] > B[0][I + 1]** THEN
  - **B[0][(I - 1) / 2] := B[0][I]**
- ELSE
  - **B[0][(I – 1) / 2] := B[0][I + 1]**
- END IF
- **I := (I – 1) / 2**
- ELSE IF **FLAG1 != 1** THEN
  - IF **B[0][I] > B[0][I - 1]** THEN
  - **B[0][(I - 1) / 2] := B[0][I]**
- ELSE
  - **B[0][(I – 1) / 2] := B[0][I - 1]**
- END IF
- **I := (I – 2) / 2**
- IF **I = 0 AND FLAG1 = 0** THEN
- **FLAG1 := 1;**
- END IF
- IF **J % 2 = 1 AND FLAG2 != 1** THEN
- IF **B[1][J] < B[1][J + 1]** THEN
  - **B[1][(J - 1) / 2] := B[1][J]**
- ELSE
  - **B[1][(J – 1) / 2] := B[1][J + 1]**
- END IF
- **J := (J – 1) / 2**
- ELSE IF **FLAG2 != 1** THEN
  - IF **B[1][J] > B[1][J - 1]** THEN
- **B[1][(J - 1) / 2] := B[1][J]**
- ELSE
- **B[1][(J – 1) / 2] := B[1][J - 1]**

- END IF
- **J := (J – 2) / 2**

- IF **J = 0 AND FLAG2 = 0** THEN
  - **FLAG2 := 1;**
- END IF

11. **END WHILE**

**Algorithm 3:** MFTS (ARR[], N)

**Input:** ARR[] is the array to be sorted, N is the size of the array ARR[]

**Output:** Sorted ARR[]

1. Define TO_LOW := N / 2 – 1
2. Define **TO_HIGH := N / 2**
3. Define **B[][] := CREATE_TOUR_TREE (ARR[], N)**
4. WHILE **TO_LOW > -1** DO
   - IF **B[0][0] > B[1][0]** THEN
     - **UPDATE (B, N, SWAP)**
   - ELSE
     - **A[TO_LOW--] := B[0][0]**
     - **A[TO_HIGH++] := B[1][0]**
     - **UPDATE (B, N, UPDATE)**
   - END IF
5. END WHILE
6. Return ARR[]

***Algorithm 1***: Is creating a tournament tree in such a way that first whole list is divided into two parts: first part contains all the biggest element or winner player by comparing adjacent elements and second part contains all the smaller elements or looser players by comparing the adjacent elements. After dividing the whole list into two parts from the winner set a looser player or smaller element among those winner players or biggest elements goes on the root of the tree and similarly from the looser set winner player among those looser players goes the root of another tree.

***Algorithm 2:*** Takes the type, if the type is update than it simply replace the biggest value with the rooted element of the winner set and smallest value with the rooted element of the looser set. After this replace this newly updated value with its actual position in both the tree. But if the type is swap than it first swap the rooted value with each other and than replace them to their actual position in both the trees.

***Algorithms 3***: First create the tournament trees by calling Algorithm 1 and pass the type to the Algorithm 2 for balancing the tournament trees. If the tournament is balanced than it passes update in the type field and if the tournament is not balance than it pass the swap in the field of type.

Some lemmas and theorems are given to prove the correctness of the algorithm and derive the running time complexity. Theorem-1 proves that the proposed algorithm requires *N/4* rounds of swapping in a list of *N* elements. In Lemma-1, it is proved that the first elements for which their position in sorted list is found are the median elements. In Theorem-2, with the help of Lemma-1 it is proved that the proposed algorithm sorts the list.

**Theorem 1:** *There are at most N/4 rounds of swapping takes place to balance both the trees and since in each round 2log(N/2) swaps are needed thus the overall maximum swaps required are (N/2)log(N/2).*

***Proof:*** If there are *N* players in a tournament then we can divide them into two sets of *N/2*players each such that one set is of top ranked *N/2* players and another set is of *N/2* bottom ranked players.

Suppose *m* players are in $N_t$ but they originally belongs to $N_b$ this implies that there are exactly *m* players in $N_b$ also which originally belongs to $N_t$. This happens if *m* players of $N_t$ plays against another *m* players of $N_t$ itself. Thus losing *m* players will go to the set $N_b$. Similarly, if *m* players of $N_b$ plays against another *m* players of $N_b$ then the winning *m* players of $N_b$ will go to $N_t$ and hence the tournament or fixture becomes imbalanced.

To balance the fixture we need to find these *m* losing players from $N_t$ and *m* wining players from $N_b$ and should be swapped with each other. Hence total rounds of swaps required are *"m"*.

Let $x_t$ and $y_t$ area mong top ranked *N/2* elements and $x_b$ and $y_b$ are among bottom ranked *N/2* elements and it should be decided which set these four player belongs to.

Assuming that $x_t$ and $y_t$ plays against each other and $R(x_t, y_t) = y_t$ then $x_t$ goes to *loser set* $N_b$ and $y_t$ goes to *winner set* $N_t$. This is shown in equation (10.1) and (10.2).

$$R(x_t, y_t) = y_t \quad \textbf{(10.1)}$$

$$x_t \in N_b \text{ and } y_t \in N_t \quad \textbf{(10.2)}$$

Also, assuming that $x_b$ and $y_b$ plays against each other and $R(x_b, y_b) = y_b$ then $x_b$ goes to *loser set* $N_b$ and $y_b$ goes to *winner set* $N_t$ as shown in equation (11.1) and (11.2).

$$R(x_b, y_b) = y_b \quad \textbf{(11.1)}$$

$$x_b \in N_t \text{ and } y_b \in N_b \quad \textbf{(11.2)}$$

From equation (10.2) and (11.2), $x_b$ is in $N_t$ instead of $N_b$ and $x_t$ is in $N_b$ instead of $N_t$, the fixture or tournament is imbalanced and should be swapped to balance it. So, after swapping $x_b$ and $x_t$ the equations (10.1), (10.2), (11.1), and (11.2) changes to equations (12.1), (12.2), (13.1), and (13.2) respectively which balances the fixture.

$$R(x_b, y_t) = y_t \quad \textbf{(12.1)}$$

$$x_b \in N_b \text{ and } y_t \in N_t \quad \textbf{(12.2)}$$

$$R(x_t, y_b) = y_t \quad \textbf{(13.1)}$$

$$x_t \in N_t \text{ and } y_b \in N_b \quad \textbf{(13.2)}$$

For a single swap, four players go into their correct corresponding sets. So, if there are *N* players then at most *N/4* rounds of swap are needed. Hence, equation (14.1) holds.

$$m = N/4 \quad \textbf{(14.1)}$$

For updating a path in both the trees *2log(N/2)* swaps are needed, then

$$N/4(2\log(N/2)) = (N/2)\log(N/2)\ldots \quad \textbf{(14.2)}$$

**Lemma 1:***If the fixture is balanced i.e. the loser among all winners in winner set $N_t$, say x and the winner among all losers in loser set $N_b$, say y then R(x, y) = x. It means that x and y are the middle ranked players when the players are sorted according to their rankings.*

***Proof:*** From the set of winners $N_t$, we find the biggest loser of that set which means that out of *N/2* players of $N_t$ this losing player *"x"* will be ranked last in the set and remaining *(N/2)-1* players have some rank above the last rank within the set. This operation fixes the position of loser *" x"* in $N_t$. Similarly, in $N_b$ we find the biggest winner i.e. the rank of this player is first among all the *N/2* players of $N_b$. This winner player's rank is fixed in $N_b$ and the remaining *(N/2)-1* players have some rank below this winning player *"y"* which fixes its position.

Given that *R(x, y) = x* which shows that the fixture is balanced, so the remaining *(N/2) -1* players other than *x* of $N_t$ has rank above all the players of $N_b$ including *"y"*. Also the remaining *(N/2)-1* players of $N_b$ other than *"y"* has rank below all the players of $N_t$ including *"x"*, and hence fixes the position of *x* and *y* making them the middle rank elements (*"y"* follows *"x"*) when all the *N* players are ranked.

**Theorem 2:** *All the players are arranged according to their ranks in ascending order on the basis of median.*

***Proof:*** Using *Lemma 1*, we found players *"x"*and*"y"* are lying in the middle (*"y"* follows *"x"*) when all the players are ranked in ascending order.

Given $x \in N_t$ and $y \in N_b$ as in *Lemma 1*, the proposed algorithm replaces *r(x)* with a rank such that *"x"* becomes the biggest winner in the set of winners $N_t$, so that no remaining *(N/2)-1* players beat *"x"*. Now if the overall $N_t$ set loser is found that will be from the remaining *(N/2) -1* players which is originally the second last player in the set $N_t$ and let the player be called *"$x_1$"*.

Similarly, the proposed algorithm replaces *r(y)* with a rank such that *"y"* becomes the biggest loser in the set of losers $N_b$, so that no remaining *(N/2)-1* players lose to *"y"*. Now if the overall $N_b$ set winner is found that will be from the remaining *(N/2)-1* players which is originally the second best player in the set $N_b$ and let this player be *"$y_1$"*.

Place *"$x_1$"* before *"x"* and *"$y_1$"* after *"y"* since we know that$r(x_1) < r(x) < r(y) < r(y_1)$

So continuing in this manner we get the sequence,

$r(x_{(N/2)-1}) < r(x_{(N/2)-1}) \ldots r(x_1) < r(x) < r(y) < r(y_1) \ldots < r(y_{(N/2)-1})$ which is in sorted manner. Hence, the list is sorted on the basis of rank of the players.

## IV. COMPLEXITY ANALYSIS

The amount of time needed by MFTS depends on the three algorithms mentioned in Algorithm 1, Algorithm 2, and Algorithm 3.In algorithm 1, two initial tournament trees are generated from *N* elements of the list which are arranged in two

corresponding arrays such that in first array $i^{th}$ element is the largest element among elements stored at $(2i)^{th}$ and $(2i+1)^{th}$ position in this array and in second array $i^{th}$ element is the smallest element among elements stored at $(2i)^{th}$ and $(2i+1)^{th}$ position in this array, assuming these arrays are indexed from 0.The former array represents the loser set $N_b$ and the latter array represents winner set $N_t$. These arrays are of the size *(N-1)* each, and are first filled from$(N-1)/2^{th}$ position to last by comparing two elements of the given list at a time, such that the larger among them goes in $N_t$ array and smaller goes in $N_b$ array. This operation takes *N/2* comparisons. And then for filling the first half of both the arrays require *(N/2)-1* comparisons each. So, total number of comparisons required by algorithm 1 is *(N/2 + N/2 – 1 + N/2 – 1)* which comes out to be *(3(N/2) – 2)*.

Algorithm 2 is used to balance the imbalanced tournament as described in definition 4 and 5.To check if the tournament is balanced, root element of both tournament trees are compared to each other, which requires exactly *1* comparison. Algorithm 2 is also used to update the tournament trees when elements are taken out from these trees to arrange in the resultant array corresponding to their respective position as per the rank. In each such updating iteration, traverse the root element to its position at the last level in the tournament tree. This takes *2log(N/2)* comparisons to go down to the last level for both the trees. After updating the rank of the positioned elements, both the trees along the traversal path is updated which takes another *2log(N/2)* comparisons. Hence, total number of comparisons required here are *4log(N/2) + 1*. Number of swaps needed to balance the tournament is *(N/2)log(N/2)*as proved in Theorem-1.

In algorithm 3, array is filled with the elements of unsorted list in ascending order. It first calls algorithm 1 to create tournament trees, then using algorithm 2 it balances the imbalanced tournament trees. Finally, it starts filling the elements in increasing order in a batch of two elements. So, if there are *N* elements then it will take *N/2* iterations to complete the filling process.

In best case, when the tournament is already balanced then algorithm 2 for balancing is never called and hence, the total number of comparisons required for sorting is shown in equation (15).

$$3(N/2)–2+N/2(4\log(N/2))+1 = 2N\log(N/2)+3(N/2)-1 \quad \textbf{(15)}$$

In worst case, the tournament is imbalanced and at most *N/4* rounds of swaps of needed which increases number of comparisons required. So, overall comparisons needed are the number of comparisons in equation (15) including these extra comparisons which gives the total comparisons as shown in equation (16).

$$2N\log(N/2)+3(N/2)-1+N/4(4\log(N/2)+1)$$

$$= 3N\log(N/2)+7(N/4)-1 \quad \textbf{(16)}$$

## V. NUMERICAL EXAMPLE

**Given list of Element:** 9 1 2 4

**Step 1:** First divide the given list into two parts as trees as mention in ***Algorithm 1***

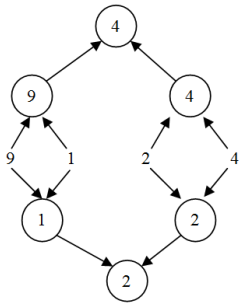



**Figure 1**

**Step 2:** check whether both the tree is balance or not by comparing root elements as mentioned in ***Algorithm 2***. Here looser among the winner (4)> winner among the looser (2) so tournament is balanced. Hence 2 and 4 are the median of the given list.

So only replace the rooted element on their actual position by biggest and smallest value

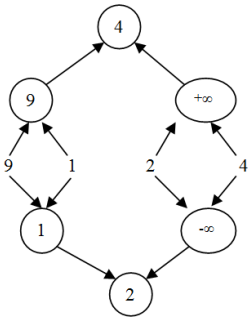



**Figure 2**

| | | | |
|---|---|---|---|
| | 2 | 4 | |

**Step 3:** Now regenerate tournament tree based on comparison on newly updated values.

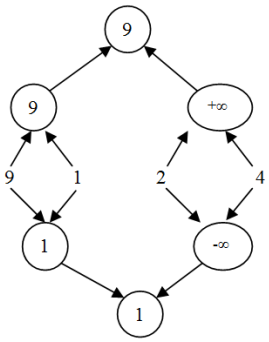



**Figure 3**

Here 9>1 so again tournament is balance so replace biggest and smallest value with actual position of both the rooted elements.

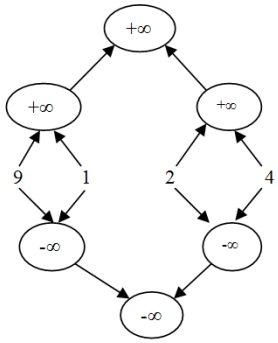



**Figure 4**

| 1 | 2 | 4 | 9 |
|---|---|---|---|

Here all the elements are replace by biggest and smallest values that means all the elements are sorted.

## VI. EXPERIMENTATION

### Test Data

The proposed algorithm's performance was compared with performance of Merge sort and Heap sort in terms of the time required to sort the list of differing size. The size of list *n* considered for evaluation was: 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, and 10000000. For each input size *n*, there were five different files of varying data redundancy in a set so as to consider best, average, and worst cases. Then the average of all five running time is calculated for each input set for evaluation.

### Implementation

The proposed algorithm was implemented in C++ along with merge sort and heap sort. Execution time was calculated on two systems with different specifications. The specifications of the systems are mentioned in Table-1 and Table-2.

**Table 1: Specifications of System-1**

| | |
|---|---|
| **CPU** | Intel i5-3470 3.2GHz |
| **RAM** | 2.0 GB |
| **OS** | Windows 8.1 x64 |

**Table 2: Specifications of System-2**

| | |
|---|---|
| **CPU** | Intel i5-3210 2.5GHz |
| **RAM** | 4.0 GB |
| **OS** | Windows 8.1 x64 |

## RESULTS AND DISCUSSIONS

Table 3 shows the average running time (in seconds) of sorting algorithms for input size *n* of 1000 to 10000000 on system-1. As shown in table-3, merge sort took 53.398 seconds of execution time for input size of 10000000 because System-1 had 2GB of RAM which was not sufficient to store the large recursion of this algorithm. Corresponding to table-3, Graph 1-a) is the bar graph for input size 1000 to 100000 and Graph 1-b) is the bar graph for input size 500000 to 10000000.

Similarly, table-4 shows the average running time (in seconds) of MFTS, merge sort, and heap sort on system-2. Graph 2-a) is the bar graph representation of the running time shown in table-4 for input size 1000 to 100000, and Graph 2-b)

is the bar graph for input size 500000 to 10000000 as shown in table-4.

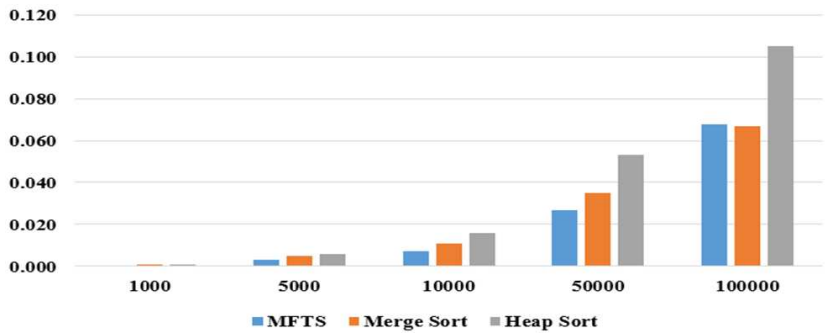
On *y-axis* of Table-3 and table-4 time in seconds is represented and on *x-axis* input size *n* is represented, and the graph is plotted against these.

**Table 3: Running Time (in Seconds) on System 1**

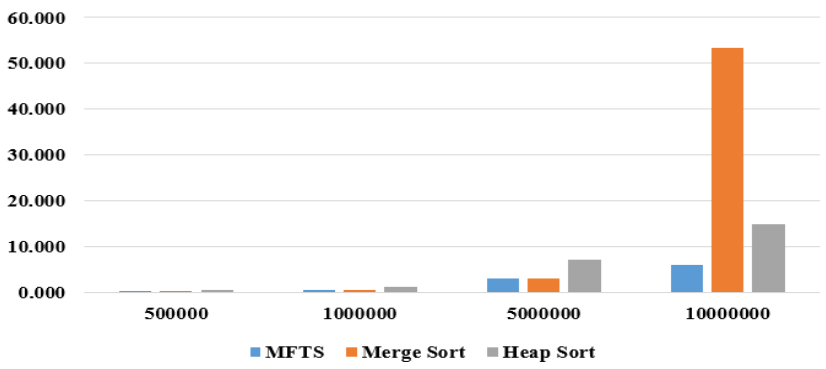
| Input Size | MFTS | Merge | Heap |
|---|---|---|---|
| 1000 | 0.000 | 0.001 | 0.001 |
| 5000 | 0.003 | 0.005 | 0.006 |
| 10000 | 0.007 | 0.011 | 0.016 |
| 50000 | 0.027 | 0.035 | 0.053 |
| 100000 | 0.068 | 0.067 | 0.105 |
| 500000 | 0.227 | 0.275 | 0.542 |
| 1000000 | 0.487 | 0.543 | 1.160 |
| 5000000 | 2.919 | 3.055 | 7.038 |
| 10000000 | 5.929 | 53.398 | 14.841 |

**Table 4: Running Time (in Seconds) on System 2**

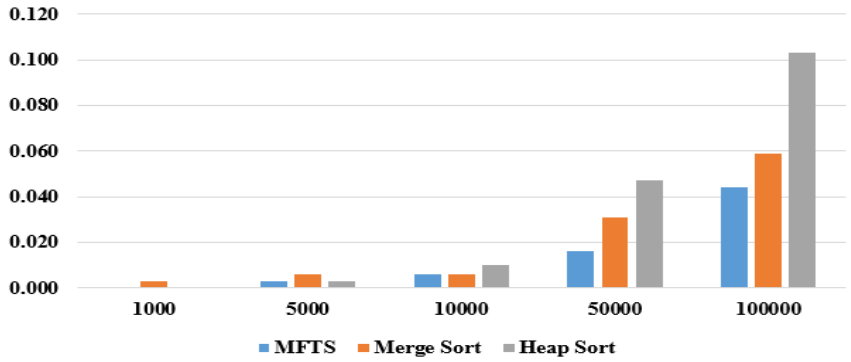
| Input Size | MFTS | Merge | Heap |
|---|---|---|---|
| 1000 | 0.000 | 0.003 | 0.000 |
| 5000 | 0.003 | 0.006 | 0.003 |
| 10000 | 0.006 | 0.006 | 0.010 |
| 50000 | 0.016 | 0.031 | 0.047 |
| 100000 | 0.044 | 0.059 | 0.103 |
| 500000 | 0.256 | 0.300 | 0.628 |
| 1000000 | 0.562 | 0.609 | 1.347 |
| 5000000 | 3.169 | 3.206 | 8.013 |
| 10000000 | 6.550 | 6.660 | 17.425 |



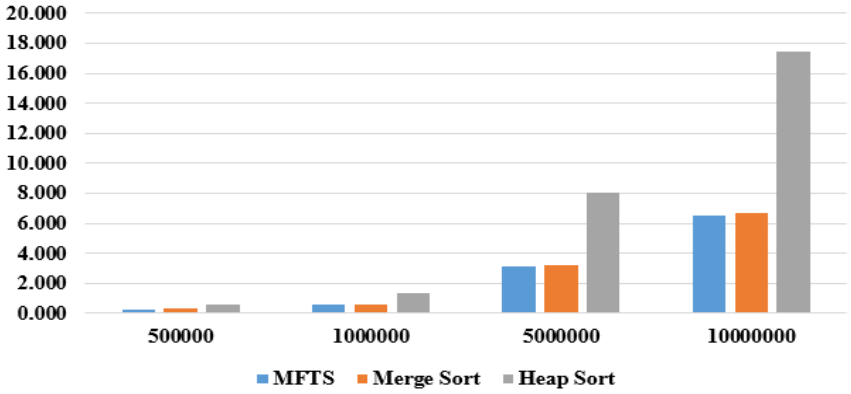

**Graph 1: a) Running Time (in Seconds) on System 1 for Input Size 1000 to 100000**




**Graph 1: b) Running Time (in Seconds) on System 1 for Input Size 500000 to 10000000**




**Graph 2: a) Running Time (in Seconds) on System 2 for Input Size 1000 to 100000**




**Graph 2: b) Running Time (in Seconds) on System 2 for Input Size 500000 to 10000000**

## VII. CONCLUSIONS

This paper proposed a new and efficient algorithm for sorting. As the above shown results in graph 1 a, b and graph 2 a, b prove that proposed sorting algorithm works very well in best as well as worst case scenario. Proposed sorting algorithm is a comparison based sorting algorithm with time bounds O (n log n) in worst case, average case and best case. Proposed sorting algorithm makes less number of swaps as compare to merge sort, heap sort, and bubble sort. Proposed sorting algorithm has been tested on large data set as shown in Table 3 and Table 4 with varying repetition ratio of elements and obtained results are compared with heap sort and merger sort algorithms, and it has given better result as compare to heap sort and merge sort algorithm as shown in graph 1 a, b and graph 2 a, b. In future work it can be implement in parallel to reduce the time complexity. Finding looser among winner set and winner among looser set can be implemented in parallel and swapping or replacing the values also can implement in parallel.

### *VIII. REFERENCES*

1. *Fabian Kirchhoff, "Modeling Delay Propagation in Railway Networks", Operations Research Proceedings 2013, ISBN 978-3-319-07001-8,PP237-242.*
2. *Max Vladymyrov, Miguel ´A. Carreira-Perpin, "Entropic Affinities: Properties and Efficient Numerical Computation", Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.*
3. *S.-H. Shiau and C.-B. Yang. A fast sorting algorithm and its generalization on broadcast communications. preprint, 18 pages, 2000.*
4. *S.-H. Shiau and C.-B. Yang. A fast sorting algorithm on broadcast communications. Algorithmica, :6 pages, 2000.*

5. *Guy E. blelloch, Leonardo Dagum, Stephen Smith, Kurt Thearling, and Marco Zagha, "An Evaluation of Sorting as a supercomputer Benchmark", NAS Report RNR-93-002, 1993.*
6. *Kurt Thearling, Stephen Smith, "An Improve Supercomputer Sorting benchmark", Proceedings of supercomputing, 1992.*
7. *Steven S. Skiena, "The Algorithm Design Manual", state university of New York, Stony brook, NY11794-4400.*
8. *Rohit Joshi, Govind Singh Panwar, PreetiPathak, " Analysis of Non-comparison Based Sorting Algorithms: A Review", International Journal of Emerging "Research in Management &Technology, ISSN:2278-9359(Volume-2, Issue-12), December 2013.*
9. *Curi-Quintal, L.F.,Cadenas, J.O, MegsonG.M.,"Bit Index Sort: A Fast Non-comparison Interger Sorting Algorithm for Permutation.*
10. *Alexander Stepanov, AaronKershenbaum, "Using Tournament Trees to Sort", polytechnic University 333 Jay Street, Brooklyn, New York.*
11. *Donald Knuth. The Art of Computer Programming, Vol 3: sorting and searching, Third edition. Addison Wesley, 1997. ISBN 0-201-89685-0. pp. 106-110 of section 5.2.2: sorting by Exchanging.*
12. *V. Mansotra, Kr. Sourabh, "Implementing Bubble Sort Using a New Approach", Proceedings of the 5th National Conference, INDIA Com 2011, New Delhi.*
13. *J. W. J Williams. Algorithm 232 –Heap sort, 1964, Communications of the ACM 7(6): 347-348.*
14. *Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0., pp. 138-141, of Section 5.2.3: Sorting by Selection.*
15. *Mrs. P. Sumathi, Mr. V.V. Karthikeyan, "A New Approach for Selection Sorting", IJARCET, Volume 2, Issue 10, October 2013.*
16. *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 2.1: Insertion sort, pp.15-21.*
17. *Shell, D.L. (1959). "A high-speed sorting procedure". Communications of the ACM2 (7): 30-32. doi:l0.1145/368370.368387.*
18. *Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford [1990] (2001). "2.3: Designing algorithms", Introduction to Algorithms, 2nd edition, MIT Press and McGraw-Hill, pp. 27-37. ISBN0-262-03293-7.*
19. *Hoare, C. A. R. "Partition: Algorithm 63," "Quick sort: Algorithm 64,"and "Find: Algorithm 65." Comm. ACM 4(7), 321-322, 1961.*